Shared Virtual Memory: Progress and Challenges
Submitted for Publication *

Liviu Iftode
Computer Science Department
Rutgers University
Piscataway, NJ 08855
iftode@cs.rutgers.edu

Jaswinder Pal Singh
Computer Science Department
Princeton University
Princeton, NJ 08544
jps@cs.princeton.edu

1 Introduction

When the idea of implementing a shared address space in software at page granularity across a network of computers—i.e. the shared virtual memory or SVM approach—was first introduced[29], few would have predicted that a decade would not be enough to exhaust its research potential or even understand its performance capabilities. The initial approach used a sequential memory consistency model, which together with the large granularity of coherence and the high software overhead of communication greatly limited performance. Progress was slow until relaxed memory consistency models breathed new life into the approach in the 1990s, starting with research at Rice University[10, 24]. Software shared memory became a very active research area, with many groups designing new protocols, consistency models and systems [10, 24, 4, 33, 37, 18, 2, 32, 26]. These groups inspired and motivated one another, building on one another's ideas to push performance higher (Figure 1).

While much of the research so far has been in protocols and their implementations, as can be seen from Figure 1, with the relative maturing of protocols the field has recently been taking on a more integrated research approach. More emphasis is put on application-driven performance evaluation, on application structuring for these systems and understanding application-system interactions and bottlenecks, on comparing with fine-grain software shared memory, on architectural support, and on software tools. We believe that these less researched areas now have the most promise for driving future advances: in understanding and reducing the still considerable performance gap between hardware and software shared memory (which should be the yardstick for protocol research), in improving the protocols and system support further, and in understanding programmability.

This paper places the multi-track flow of ideas and results so far in a framework, with the goal of identifying the lessons learned so far and the major remaining challenges. The paper is divided into two major sections. The first is devoted to summarizing the development of protocols, consistency models and architectural support for shared virtual memory, and the results obtained in the literature for performance. It identifies key protocol questions that are still outstanding for further evaluation. The next section focuses on the areas of recent interest that have been generating new types of research in software shared memory, and that make this still a very interesting field of research with many open questions.

2 Consistency, Protocols and Support

A memory consistency model defines constraints on the order in which memory accesses can be performed in shared memory systems. It influences both programming complexity and performance. A strict consistency model like sequential consistency [28] is intuitive for a programmer. However, with the large granularity of coherence (a page) in shared virtual memory systems, coherence operations can be very frequent under this model due to false sharing (two processors access unrelated data that happen to be on the same page, and at least one of them writes the data). Since coherence operations are also in software and hence expensive, the performance of SVM systems under SC is very poor.

Relaxed memory consistency models allow the propagation and application of coherence operations (e.g. invalidations) to be postponed to synchronization points, greatly reducing the impact of false sharing and the frequency of coherence operations. Among the many relaxed consistency models proposed for hardware-coherent systems, release consistency [14] which separates acquire from release synchronization was the first to inspire a major breakthrough in the performance of software shared memory systems [10] (see Table 1[1]). Under release consistency and models inspired by it, processors can thus continue to share a page without any communication of data or coherence information until a synchronization point.

A memory consistency model specifies when coherence operations and data *need* to become visible. However, it

* Technical Report DCS-TR-344 Department of Computer Science, Rutgers University and TR-552-97 Department of Computer Science, Princeton University

[1] Some speedup numbers in the tables in this paper are currently approximate, since they were extrapolated from graphs.

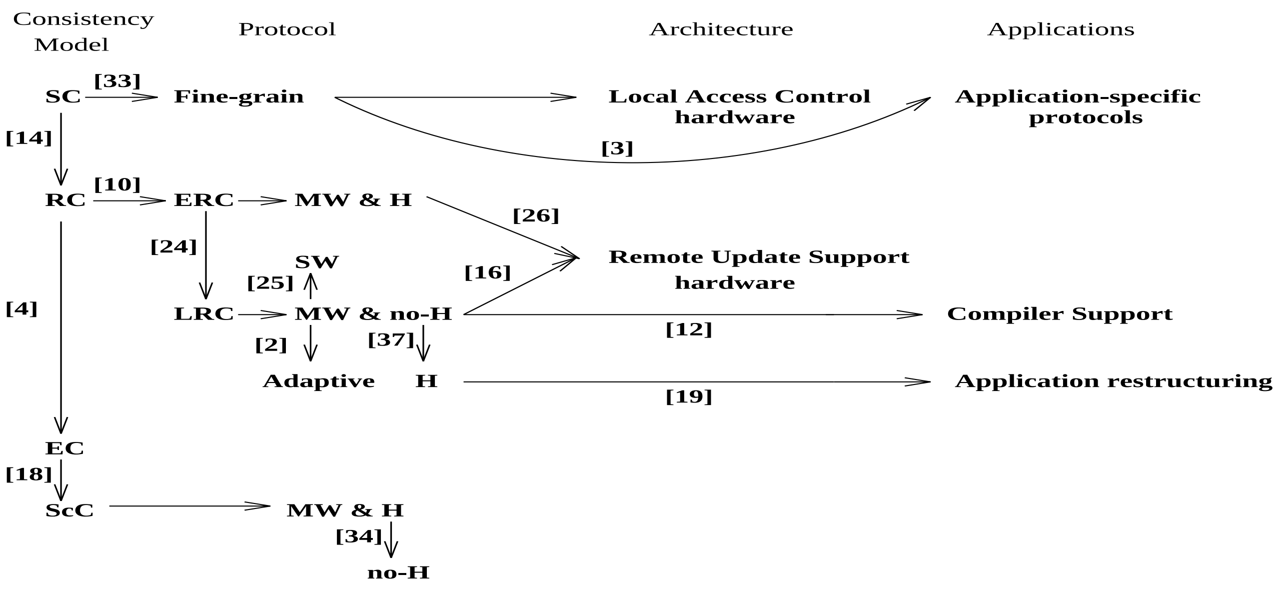

Figure 1: Research in SVM. The number in brackets next to an arc is a reference to the paper that created that arc. The figure treats LRC and ERC as protocols implementing an RC consistency model, though they may be thought of as different models. MW is multiple writer, SW is single writer, and H means home-based. SC, RC, EC and ScC are the sequential, release, entry and scope consistency models.

Benchmarks	SC	ERC
MULT	14.5	14.6
DIFF	8.4	12.3
TSP-C	11.3	12.6
TSP-F	4.7	6
QSORT	4.1	8.9
FFT	0.1	8.2
GAUSS	5.1	8.6

Table 1: Speedups for SC vs. ERC, an early implementation of release consistency for SVM in the Munin system. 16 processors [9].

can actually be implemented with various degrees of "laziness" in the propagation and application of both coherence and data. Greater laziness typically implies greater complexity and protocol state, but fewer communication and protocol operations. For example, hardware-coherent systems that use relaxed consistency tend to propagate coherence operations immediately, thus simplifying the data structures that need to be maintained for coherence (e.g. directories). On the other hand software systems based on relaxed consistency can afford to manage more complex protocol state, so to achieve better performance they often delay propagation/application until the point permitted by the consistency model.

During the past six years many SVM systems based originally on release consistency have been developed along different but related paths, with different degrees of laziness and performance tradeoffs. Most published papers used the previous protocol along their path as the base case for comparison. There is a lot of very useful performance data available in the literature, and a lot of light has been shed on protocol tradeoffs. However, the generality of the results has often been limited by two factors. First, they have typically not used a comprehensive enough set of real applications that exercise different relevant characteristics. The earlier studies in particular focused mostly on simple kernels. Second, they have made assumptions about important lower-level implementation mechanisms, as well as about architectural support and performance characteristics, which can make a great difference to the higher-level tradeoff being investigated. The effects of the these factors compared to (or as relevant to) those of the higher-level tradeoff have not been well explored. By understanding the evolution of protocols in a framework ranging from high-level consistency models down to implementation mechanisms, we can identify the most promising protocol candidates and their tradeoffs, as well as the key outstanding tradeoffs in protocols, implementation mechanisms and architectural support that continue to demand comprehensive performance evaluation. The rest of this section describes this evolution, together with available results from the literature to indicate what is known so far. In particular, the main issues we consider in this evolution are laziness in the propagation of coherence operations (e.g. invalidations), laziness in applying coherence operations, multiple writer support, and architectural support to accelerate the protocols.

2.1 Laziness in Propagating Coherence Information

Protocols that take advantage of release consistency vary in the laziness they use in various aspects (which can in fact end up changing the consistency model a little). One important type of laziness in a protocol is measured by how much it delays the propagation of coherence operations (e.g. invalidations). In eager release consistency (ERC) [10], invalidations are propagated to all sharing processors (and in fact applied there) before the issuing processor completes its next release operation. In lazy release consistency (LRC) [24], the invalidation corresponding to a write by processor A is conveyed to a processor B on demand upon B's next acquire synchronization operation, if that acquire is causally after the release by the writing processor A [2]. Munin [10] and TreadMarks [22] from Rice University were the first SVM systems to propose and implement ERC and LRC protocols, respectively.

Laziness has implications for the amount and lifetime of the buffered coherence state that must be maintained. Since ERC propagates coherence information at the release to all processors that might need it, the information can be discarded by the releaser thereafter. The only tracking data structure needed is a directory similar to the one used in hardware cache coherence. In LRC coherence information is propagated at an acquire operation only to that acquirer, and it cannot be discarded at a release since there is no way to predict which processor will issue a corresponding acquire and when. Also, the releaser has to provide to the next acquirer all the coherence information that the releaser has seen in its past but the acquirer has not. This transitive propagation of the coherence information requires an LRC protocol to store much more complex state for much longer: timestamp vectors to keep track of the causal dependencies among synchronization events, and write-notices for the coherence information received from other nodes during past synchronizations (for possible transmission in the future) [22].

Compared to ERC, LRC amplifies the advantages of relaxed consistency: fewer messages (for instance if a lock is reacquired many times by the same node, there is no communication) and potentially fewer page misses due to false sharing (if false sharing occurs between nodes A and B between the time a lock is released by node A and the time the lock is acquired by node B, then in LRC page faults will not be incurred in that period if there is no other synchronization between node A and node B during that period). On the other hand, because LRC requires propagating the transitive closure of coherence information along a causal chain, processors may receive unnecessary coherence information for which they are simply transit sites. Keeping track of the transitive closure is also more expansive in terms of protocol and memory overhead than in ERC.

Table 2 reports the results of a comparison of a particular ERC protocol similar to Munin and the TreadMarks LRC protocol [23]. The comparison is not purely about this issue, however, since there are other differences: This ERC protocol uses a hybrid update-invalidation approach discussed later, while the LRC protocol is invalidation-based (both are also multiple-writer protocols, which we will discuss in Section 2.3). The LRC protocol seems significantly advantageous for several programs, except for one (FFT) in which the update nature of the ERC protocol reduces the page fault latency.

[2] ERC and LRC are quite similar as reasoning models for a programmer, but there are programs for which they can produce different answers and they are thus different consistency models.

Benchmarks	ERC	LRC
SOR	7.2	7.4
TSP	6.1	7.2
Water	3.8	4.6
Barnes	2.6	3.2
FFT	4.2	3.6
IS	4.4	5.8
ILINK	5.9	5.8
MIP	4.2	5.6

Table 2: Speedups for particular ERC vs. LRC protocols. 8 processors [23]

2.2 Laziness in Applying Coherence Information

While a protocol may *propagate* coherence information (e.g. invalidations) eagerly at a release, as in ERC above, various degrees of laziness can be used for actually *applying* or performing the coherence operations and in processing acknowledgments. These are often glossed over in comparing eager propagation protocols with LRC, but they can have significant performance effects.

Delayed consistency(DC) [11] is an example of eager propagation but lazy application.[3] In this case, incoming invalidations are not applied right away but are queued at the destinations until they perform their next acquire operation (whether causally related to that release or not). Acknowledgments are sent back as soon as invalidations are received so that the releaser can proceed. It is also possible to reduce release latency by queuing the incoming acknowledges at the releaser, and allowing it to check for them only at the next acquire request it receives (for any synchronization variable). With these queuing schemes, a DC protocol avoids the false sharing misses which may have occurred due to accesses between the time the invalidations are received and the next acquire operation (a typical example is the next barrier), though it is different from LRC in that it does not postpone invalidation application until the next causally related acquire. The queue mechanisms needed by DC can be used in hardware protocols as well as in software DC protocols [11, 38].

Two RC-based protocols that allow only a single writer to a page at a time (see next subsection) but propagate and apply invalidations at different times have been compared with SC on a number of SPLASH-2 applications that cover most of the basic sharing patterns [38] One is an eager-propagation but lazy-application delayed consistency protocol as described above, and the other is a single-writer LRC protocol (i.e. lazy propagation and application) similar to the one proposed in [25]. Table 3 shows some of the speedup comparisons, obtained on the Wisconsin Typhoon-zero platform. The lazier application and propagation of the LRC protocol have some advantages, especially in complex irregular applications that use substantial lock synchronization. The study is limited by

[3] One may call all eager propagation protocols ERC protocols even though they implement somewhat different consistency models, or one may label ERC to be eager propagation and eager application. We try to do the latter consistently.

the fact that it uses hardware access control and by some performance features of the platform.

Benchmarks	SC	DC	LRC
LU	8.6	8.6	8.4
Ocean	2.7	3.8	5.7
Water-Nsquared	11	11.3	11.3
Volrend	0.8	1.7	2.9
Water-Spatial	4.9	5.8	7.3
Raytrace	6.6	8	9
Barnes	0.9	1.9	2.2

Table 3: Speedups for Single Writer Protocols: SC vs. DC. vs LRC. 16 processors [38].

In contrast, the Munin system discussed earlier implements an eager-propagation, eager-application (ERC) protocol, in which invalidations or updates are performed as soon as they arrive (i.e. it is not DC) and a releaser waits for acknowledgments before proceeding. Release latency is large due to large round-trip message cost and remote interrupt overhead, both of which are on the critical path. The Cashmere system from Rochester [27] proposes a lazy-application, eager-propagation protocol that queues invalidations but still waits for acknowledgments at release time. Overall, the performance-complexity tradeoffs in laziness of propagating and applying coherence are not yet very clear.

2.3 Multiple-writer Protocols

Using relaxed memory consistency alone helps alleviate read-write false sharing between synchronization points: One processor can keep writing a page locally while other processors read it. However, if we want to avoid the immediate propagation of coherence when different processors write the same page between synchronization points (multiple writer or write-write sharing), this takes more effort and protocol complexity. The writes performed by different processors must be kept track of and merged before another processor needs to see them according to the consistency model.

The first multiple-writer mechanism was used in Munin itself, but its potential was better exploited with the LRC protocol in TreadMarks. In this multiple-writer scheme, at a release point a writer determines the changes it has made to a page—by computing the differences (diff) between the version of the page at the beginning of the synchronization interval and the version at the end (i.e. just before the release)—and it holds that diff locally. When a processor incurs a page fault it has to obtain the diffs from the relevant writers and merge them into the faulting copy of the page in the right order to bring it up to date.

While ingenious, this original multiple writer scheme has some potential problems. First, when there is indeed write-write false sharing, a processor will need to get diffs from *multiple* writer nodes and apply them in the appropriate order. This can lead to a large number of messages that while the can be overlapped to some extent are in the critical path, increasing page fault latency. Second, the need to keep diffs around at the writers can generate very large memory consumption (often larger than the application data set size itself [37]), thus greatly limiting scalability; periodic garbage collection is needed, which adds more overhead. Third, with a migratory sharing pattern diffs can accumulate rapidly because each process that acquires the lock fetches the diffs and applies them in order, but must also retain them all them so they may be passed on to the next acquirer processor (due to the need for satisfying causality). This can generate substantial communication traffic in addition to memory consumption. Finally, diffing requires several memory-intensive operations (page copy to create a clean version of the page at the beginning of the synchronization interval, page to page comparison to compute the diff, and applying the diff upon a page fault). In addition to their direct costs, these operations substantially pollute the first-level cache and hurt performance further. These problems have prompted the development of new solutions for multiple writers which are examined in the next subsection.

2.3.1 Alternative Multiple Writer Schemes

One approach which can alleviate many of the problems with the original LRC multiple-writer scheme is to use a home-based multiple-writer scheme. In an all-software home-based LRC protocol (HLRC), evaluated in [37], a home node is selected for each shared page (usually the first writer of that page). Diffs are eagerly propagated to the home at a release, and then discarded at the writer. At the home, diffs are immediately applied as soon as they are received, then discarded there too. In this way the home copy of the page is kept up to date in accordance with the consistency model. On a fault, a non-home processor can obtain the whole page from the home in a single round-trip message, handling write-write false sharing without needing to go to multiple sites for diffs (the first problem above). The performance disadvantage of the home-based multiple-writer scheme is that the whole page is transferred on a fault even if only some of it has been modified, and that performance can be poor for some applications if homes are not assigned properly. The performance advantages are (i) the reduction in the number of messages needed on a page fault when there is write-write false sharing, (ii) the fact that since diffs are applied only once at the home faulting processors do not have to worry about the proper ordering of diffs, and (iii) the fact that writes performed at the home simply update the page and do not trigger diff computation. This last feature can make a big difference to performance for applications in which a page has only or mostly a single writer: If that writer becomes the home then no diffs are computed and the only communication is the page transfers themselves.

Propagating changes eagerly to the home makes diff lifetimes extremely short, thus eliminating the second, memory overhead problem of the original approach and the need for garbage collection. Also, a chain of diffs does not have to be maintained and passed around to preserve causal order (e.g. in migratory sharing), since the whole updated page is communicated from the home on demand. Tables 4 and 5 compare the home-based LRC (HLRC) with a single-writer LRC and with an original

Benchmarks	SW-LRC	HLRC
LU	8.0	7.9
Ocean	6	8.7
Water-Nsquared	11.6	11.8
Volrend	3	9
Water-Spatial	7	12
Raytrace	9	13
Barnes	3	6

Table 4: Speedups for Single Writer vs Multiple Writer Home-based LRC on Typhoon-zero. 16 processors [38].

Benchmarks	LRC	HLRC
LU	11.5	13.9
SOR	13	22.7
Water-Nsq	11.7	18.9
Water-Spatial	14	20
Raytrace	10.6	26.8

Table 5: Speedups for non home-based multiple-writer LRC vs home-based multiple-writer LRC on Intel Paragon. 32 processors [37].

non-home LRC, respectively. Multiple-writer protocols are clearly very advantageous for irregular applications that tend to exhibit write-write false sharing and lock-based synchronization, and of course not advantageous for applications whose patterns of page access are mostly single-writer [25]. And at least on the Paragon platform, the home-based software multiple-writer protocol seems to often substantially outperform the original, non home-based one. A simulation-based comparison of home-based with architectural support versus non-home-based LRC can be found in [17].

Two recent papers [2, 21] independently propose schemes to improve migratory sharing (the third problem above) by recognizing this sharing pattern and treat it differently. For pages that exhibit migratory sharing and not write-write false sharing within a synchronization interval, there is clearly no need for a multiple writer protocol. A protocol that simply passes on the whole page with ownership each time the lock is transferred (using a single-writer protocol for the page, just as in hardware cache coherence) will save storage, will not require diffs to be computed or carried around, and will likely perform better. These proposals use adaptive protocols: a single-writer protocol for pages detected at runtime to exhibit migratory sharing, and a TreadMarks-like multiple writer protocol for others. Note that due to its static assignment of homes to pages, a pure home-based protocol does not handle migratory true sharing quite so well. It will generate one and a half roundtrip messages for each migration (an eager propagation of the diff from the last writer to the home plus a roundtrip to get the page from the home to the next writer), compared to one roundtrip message in the single-writer protocol (directly from the last writer to the next). While adaptive protocols clearly outperform the traditional multiple-writer LRC (Table 6), the comparison with home-based protocols is not clear and must be evaluated experimentally[4]: Adaptive protocols treat migratory sharing better, while home-based protocols treat write-write false sharing better. Moreover, it will be interesting to see if an adaptive, home-based protocol can be designed.

Benchmarks	LRC	A-LRC
IS	1.2	2.1
3D-FFT	3.4	4.2
SOR	6.7	6.8
Water	3.4	3.5
TSP	5.8	5.2
Shallow	5.1	5.6
Barnes	3.8	3.9
Ilink	5.1	5.1

Table 6: Speedups for LRC vs adaptive LRC (Rice, non home-based). 8 processors [2].

Finally, to eliminate the overhead of diffs we can use other methods to track and merge modifications in multiple-writer protocols. One alternative is to maintain per-word dirty bits in software, as introduced in the Midway system [4]. This requires instrumenting memory operations in the program, incurring some runtime overhead. It has been shown to perform slightly better than diffs for migratory sharing patterns. However, the only comparison in the literature shows that overall a TreadMarks-style LRC protocol with diffs performs better than one with software dirty bits due to the cost of instrumentation [1]. The other alternative to incurring software diffing overhead is to use hardware support for either computing and applying diffs [5] or for propagating writes to the home in hardware in home-based protocols, thus eliminating diffs altogether [16, 27] (see Sec 2.4).

2.3.2 Laziness in Diffing

As with coherence propagation, there many degrees of laziness are possible in diff propagation and application. These must be clearly understood in interpreting performance results that compare protocols, since they too can influence the results significantly. In a non-home LRC protocol like TreadMarks, diffs too are propagated lazily (at a page fault). However, they can be computed with different degrees of laziness: at release time, at acquire time or at page fault time. Less lazy coherence protocols allow fewer reasonable options for diff computation. For example, in a home-based LRC, diffs should be computed no later than the acquire time, to allow enough time for diff propagation and application at the home before a page fetch occurs. By being less lazy in computing diffs, home-based protocols may compute more diffs.

An ERC protocol is even more constrained: Diffs are computed and propagated at a release together with invalidations. Otherwise, retrieving them upon a page fault may unnecessarily increase the read miss penalty since they can be retrieved only from the originating writers and not from the home. Early ERC protocols like

[4] [21] reportedly contains such results for one type of home-based protocol, but the final version of the paper is not yet available

in Munin use a hybrid update-invalidate scheme in which diffs are eagerly propagated on a release to all active copies in the sharing list, and inactive copies are invalidated. In [23] Keleher describes a different hybrid scheme for ERC and uses it to compare with TreadMarks LRC (this was the comparison shown for ERC versus LRC in Table 2: On a release, invalidations are sent eagerly to sharers as usual; on receiving invalidations, the sharers that have also modified their copies send back their diffs to this releasing processor, bringing it up to date. A subsequent page fault fetches the page only from this last releaser, just as home-based protocols fetch it from the home.

2.4 Architectural Support

The architectural support provided can also greatly impact the design choices for protocols and the results of performance comparisons, as of course can the values of architectural performance parameters. These effects and their influence on protocol comparisons are currently not very well understood.

2.4.1 Broadcast and Multicast

Interconnects that support broadcast or multicast may make an ERC protocol more attractive, since invalidations (and even diffs in an update protocol) are multicast to sharers on a release. Eager protocols may then outperform lazy protocols. For example, the protocols in the Cashmere home-based ERC system use the multicast support provided by the DEC Memory Channel interconnect to handle directory updates and synchronization [26].

2.4.2 Fine-grain Remote Operations

Support for fine-grain remote operations in the network interface may also affect some protocol design decisions. For example, home-based protocols were first proposed assuming hardware support for propagation of fine-grained writes [16, 27]. One such mechanism is called automatic update. With automatic update, writes to pre-mapped shared pages are snooped and transparently propagated to the home copy of the page if it is remote. This allows a home-based multiple writer scheme without diffs, though with potentially larger traffic on the memory bus and interconnect (since shared pages are mapped as write through in the caches so that the writes to them can be snooped on the bus). Support for AU was provided through a snooping device on the memory bus connected to the SHRIMP network interface [8].

Preliminary results of an home-based LRC implementation with AU support (AURC) on a 16-node SHRIMP multicomputer indicate that AURC typically outperforms all-software home-based LRC as long as the traffic is low or the automatic updates are performed to consecutive addresses and can be combined by the network interface. For applications like Radix sorting with large communication per computation ratio and scattered remote writes, AURC is significantly affected by contention and its performance is considerably lower than the performance of a diff-based HLRC. A study investigating automatic update support with commodity network interfaces like Myrinet finds that the performance benefits of AU support are smaller than with customized network interfaces [6].

The DEC Memory Channel also offers support for remote writes, but via explicit operations on the I/O bus [15] so writes need to be instrumented for this purpose. Despite this instrumentation overhead, the home-based Cashmere protocol implemented on a cluster of DEC workstations interconnected with Memory Channel is reported to perform very close to the performance of the diff-based original TreadMarks LRC (see Table 7)

Benchmarks	LRC	ERC
SOR	26	18
LU	12	4
Water	18	22
TSP	26	25
Gauss	10	8.9
Ilink	9	6
EM3D	10	10
Barnes	5	2

Table 7: LRC(TreadMarks) vs. ERC(Cashmere). 32 processors [26]

The Cashmere protocol uses eager coherence propagation with directories, so the design had also relied heavily on hardware support for remote reads (e.g. to access directory information efficiently), which was not available in their Memory Channel interface. The authors state that this is one of the factors which can explain the significant differences between the prototype results and results obtained in simulations that assume hardware remote read support, illustrating the dependence. Recall that other studies support the advantage of home-based protocols both with and without AU support over original TreadMarks LRC [37, 17].

2.4.3 Communication Parameters

In addition to the above forms of support, comparisons between protocols are also affected greatly by the communication mechanisms and parameter values used, and studies should examine these effects as well. In a typical SVM implementation (without a dedicated protocol processor), incoming requests interrupt the compute processor and are handled by it. The interrupt overhead is the most significant factor in the performance and scalability of a SVM protocol [7]. Using polling (e.g. at back-edges) as an alternative to interrupts in handling remote requests can improve performance or not depending on the ratio of interrupt cost vs. polling overhead on a particular platform. The overhead of interrupts or message handling in general can affect the tradeoff between delayed consistency and an eager-application ERC: If this overhead is high, then the advantage of queuing rather than immediately performing invalidations will be smaller since the overhead dominates and must be incurred anyway. This overhead also effects the tradeoff between LRC and ERC: Higher handling costs favor LRC since LRC reduces the number of messages.

To summarize, there has been a lot of excellent systems work in the last few years in developing protocols and systems for shared virtual memory. The development has led to a few key protocols and implementation mechanisms that are most promising for the future and whose tradeoffs are yet to be understood. To evaluate these tradeoffs, the standard of performance evaluation now needs to be stepped up. High-level tradeoffs among protocols are clearly quite dependent on lower-level implementation mechanisms used (e.g. to allow more laziness) as well as on architectural support and system parameters, and evaluation studies should now take these into account. Finally, a significant limitation of most of the studies performed in this area has been with applications. "Standard" applications that are used for hardware-coherent shared memory have only recently begun to be used, and most programs that have been used have been kernels with mostly very simple behavior. Performance evaluations in software shared memory should use a wider range of applications in different classes with regard to sharing patterns at page-grain [17].

Overall, we believe that while some of the open high-level and lower-level protocol questions discussed above are important to answer, even effort on improving protocols should now be spent primarily based on bottlenecks discovered in applications or insights into exploiting architectural support. The gap between hardware cache coherence and software shared memory is still quite large (e.g. [17]), even at relatively small scale. Some key bottlenecks that occur in irregular applications have been identified—for example the high cost of lock-based synchronization and dilation of critical sections due to page faults, and the fact that pages are often uselessly invalidated due to intervening synchronization even though they are write-only (e.g. images written in graphics programs)—and protocols should be adapted to alleviate these and others that are discovered.

3 Current and Future Directions

The relative maturity of protocols and some understanding of architectural support has already led to new areas for software shared memory research that have recently begun to be investigated. The time is now ripe to focus research attention primarily on the gap between the performance of software shared memory and hardware-coherent shared memory [17], and how it might be further alleviated. Some of the key directions for this are research in (i) comparing page-grained SVM with recently re-popularized fine-grained software shared memory approaches that don't rely so heavily on relaxed consistency models (to determine which is more promising for future systems), (ii) application structuring as well as protocol enhancements driven by application-protocol interactions (including work on system software support and tools), and (iii) using much more widely available multiprocessor nodes to reduce the frequency of software involvement. We also need to understand the role of hardware support, and the potential for scalability to large processor counts (not well expored yet). Finally, there have been recent efforts in relaxing memory consistency models further without increasing programming complexity too much, and while these are much less mainstream we shall mention them as well.

3.1 Fine-grain versus Page-grain DSM

The large, page-sized coherence granularity is a major performance problem in SVM systems. Although it can be substantially alleviated by using relaxed consistency models and lazy protocols, and works quite well for coarse-grained applications, for applications which exhibit fine-grain sharing these optimizations are not enough to achieve acceptable performance. Also, the fact that complex protocol activity is performed at synchronization points makes frequent synchronization a critical bottleneck. The alternative approach is to build software DSM systems with coherence performed at fine (or variable) grain. In these cases, access control and coherence are provided by instrumenting memory operations in the code rather than through the virtual memory mechanism [33].

The advantage of the fine-grained approach is simplicity, since it can use a sequential consistency model without suffering much from false sharing and communication fragmentation (and hence can also keep synchronization protocol-free and much less expensive). The programming model is exactly the same as in the more mainstream hardware DSM. In addition, tools for code instrumentation operate on executables rather than on source code, which represents a tremendous advantage over SVM. However, code instrumentation is not always portable and adds overhead on each load and store, and since communication may be more frequent the approach relies on low-latency messaging.

While Blizzard-S [33] was the first all-software fine-grain DSM, the recent Shasta system breathed new life into the approach by demonstrate quite good performance (Table 8) on a high-performance configuration (a cluster of 275 MHz DEC Alpha systems interconnected with Memory Channel) [32]. This was possible due to a number of subtle optimizations in instrumentation, some of which are specific to RISC (and even to Alpha) architectures. Among protocol-level optimizations, support for multiple coherence granularities on a per data structure basis was found to be particularly useful. It helps optimize the prefetching effect of large granuarlity communication without inducing false sharing, though it does rely on some support from the programmer.

Benchmarks	Shasta
LU-Cont	4.8
Water-Nsq	4.2
LU	3.8
Barnes	3.4
Volrend	2.8
Water-Spatial	2.0

Table 8: Shasta 8 processors [32]

Fine-grained approaches and SVM are two very different approaches to software shared memory, but there is no reported comparison between them. However, a group of researchers compared the two approaches on a fairly large and varied set of applications using the Typhoon-0 platform, which allows protocols to be run in software but uses a uniform *hardware* access control mechanism for both approaches [38]. The results show that for almost all applications, the page-grain home-based LRC protocol performs similarly to or better than a fine-grain SC protocol (even though there is no instrumentation overhead), except in one case (the original Barnes application before restructuring) where there is an overwhelming amount of synchronization (Table 9). However, this result for a particular, quite dated platform, and does not use full software access control. More research is needed to understand the tradeoffs between the two approaches, and to determine whether one is clearly superior to the other given future trends in communication performance.

Benchmarks	FG- SC	HLRC
LU	6.0	7.9
Ocean	6.1	8.7
Water-Nsquared	12.6	11.8
Volrend	6	9
Water-Spatial	12.7	12
Raytrace	14	13
Barnes	7	6

Table 9: Fine-grain SC vs HLRC. 16 processors [38]

3.2 Applications and Application-driven Research

Another major area enabled by SVM progress is research in applications for these systems. Even when "standard" applications have been used to evaluate SVM tradeoffs, the versions used have been those that were written for hardware coherence. Structuring the applications or data structures more appropriately for SVM might change the performance dramatically [19]. Simultaneous research in applications and systems will help us understand whether SVM can indeed work well for a wide range of applications, how applications might be structured to perform portably well across different shared memory systems, and what types of further protocol optimizations and architectural support might be useful.

Understanding application-protocol interactions more deeply can help in two ways: improving protocols to match the applications better, or (ii) restructuring applications to interact better with the protocols. In the former area, early work done in application-specific protocols for fine-grained software shared memory showed fairly dramatic performance gains [3]. In the SVM context, research has been done in having the compiler detect mismatches between application sharing patterns and protocols and adjust the protocol accordingly (or the user may indicate the sharing patterns to the system) [12]. For example, the compiler may detect write-only pages and prevent the SVM protocol from invalidating them at synchronization points, or it can indicate to the protocol which data should be eagerly propagated at a barrier in order to reduce the following read miss latency.

For restructuring applications, a study was performed to examine application restructuring for SVM systems and the performance portability of applications across SVM and hardware-coherent systems [19]. The study found that in most cases the applications can indeed be restructured to deliver good performance on moderate-scale SVM systems, the improvements in performance being quite dramatic. For example, since synchronization is particularly expensive due to protocol activity, parallel algorithms can be used that trade some computational imbalance for reduced synchronization frequency. However, the improvements needed are usually quite substantial, far more toward algorithmic changes than simple padding and alignment. Fortunately, most of the changes are performance portable to hardware-coherent systems, though with much smaller impact. More research is needed in developing guidelines for structuring applications in performance-portable ways. Another very important area is the development of tools for understanding performance bottlenecks in applications, since many problems arise due to contention which is very difficult to diagnose, as well as correctness tools for checking violations of consistency.

3.3 SVM across SMP Clusters

Another recent development in software DSM is motivated by the increasing popularity of small-scale hardware-coherent multiprocessors, and the development of "systems of systems" or clusters that use these multiprocessors as their nodes. There are two strong arguments for using a software DSM layer across the nodes in such a cluster. First, it provides a uniform (coherent shared memory) programming interface rather than a hybrid message-passing/shared-memory interface. Second, because the local coherence (within the multiprocessor) is performed in hardware, the overall performance of the system is expected to be better than the performance of the DSM on uniprocessor clusters, due to less frequent software involvement in interprocessor communication. Software shared memory may be a good way to extend this attractive programming model from moderate-scale hardware-coherent systems built in industry to much larger systems of systems.

Building an efficient DSM on a multiprocessor cluster is not as simple as porting a uniprocessor protocol. Issues like laziness, multiple writers and protocol handling must be revisited. With respect to laziness, the question is how quickly does the coherence information from outside propagate within the multiprocessor. For example, when one processor has to invalidate a page, should the page also be invalidated for other processors in that node [30, 35]? There is a tension between the need for some software involvement even within a node to provide laziness, and the need to reduce software eliminate software involvement to exploit hardware sharing within a node. For multiple-writer schemes, the question is how to prevent other processors in the same node from writing into the page while one processor fetches the page as a result of an invalidation. One of the first implementations indicates that shooting down the TLBs inside the multiprocessor can be expensive particularly if it occurs frequently [13]. An alternative scheme called incoming diffs was suggested in [35], but found to not help very much. For protocol handling, there are questions about whether to use a dedicated protocol processor within the node, and if not, how to distribute protocol processing in response to incoming messages across processors [20]. Systems have also been developed to implement SVM across hardware DSM rather than bus-based machines [36].

The performance gains in switching from uniprocessor to multiprocessor clusters are not yet clearly understood, and may be affected by the performance of the communication support within and across nodes, in addition to the degree of laziness implemented in the protocol. In the only two published papers describing real software DSM systems built across SMP clusters [31, 35] the comparison is made between the real DSM across multiprocessor clusters and a DSM for uniprocessor nodes emulated on the same hardware. This means that the differences which would have occurred in terms of bus and network contention between these two approaches are not exposed. At the same time, simulation studies indicate that while there is usually a benefit to using multiprocessor rather than uniprocessor nodes [20, 6], the benefit is usually small when node size is small relative to overall system size.

3.4 Beyond Release Consistency

Beyond release consistency the consistency model performance game is much more difficult to play. Protocols and coherence state become too complicated to be implemented in hardware. Software solutions too may require additional hardware or compiler support. Worse still, programming complexity also increases. While early results did not indicate significant performance benefits, recent results are somewhat more encouraging.

Entry Consistency [4] proposed the idea of binding data to synchronization variables, and at a synchronization event making only the data bound to that variable coherent. However, the programmer had to provide the binding, which was a major burden. Scope consistency (ScC) [18] shows that the effect of the synchronization to data binding can be achieved without including the binding requirement in the definition. Instead, the data to lock (scope) association is achieved dynamically when a write access occurs inside a scope. This reduces the programming complexity dramatically, although there are programs where more effort is needed beyond programming for RC. By still being page-grained, unlike EC ScC is able to reduce false sharing while still preserving the prefetching effect of pages, and while keeping the state overhead required low.

ScC was initially evaluated through simulation with a home-based protocol using automatic update support [18], but in the same paper an all-software home based ScC protocol is also proposed. The model has more recently been implemented for a non-home based multiple-writer scheme similar to that used in TreadMarks, in the Brazos protoype [34]. This implementation relies heavily on the broadcast support of the interconnect (100 Mbps Ethernet) for a hybrid update-invalidate protocol, but it corroborates the promise of the earlier simulation results (Table 10). While the programming burden still keeps these models outside the mainstream, EC may fit well with object-oriented languages like Java which inherently provide for object-to-lock association, and ScC's small programming burden may be alleviated by tools to check if a properly labeled program for RC also satisfies ScC.

Benchmarks	LRC	ScC
SOR	5.5	5.61
ILINK	2.3	3.4
Barnes	2.2	4
Raytrace	0.5	1.8
Water	3.8	3.9

Table 10: Speedups for LRC vs ScC. 8 processors [34]

4 Conclusions

There has been a lot of progress in shared virtual memory over the last several years. Protocols and their implementations have been improved steadily, fueled by relaxed memory consistency models, and are now at a stage of relative maturity. Understanding some critical tradeoffs among the more recent protocols requires further application-driven evaluation using a much wider range of applications and architectural parameters than before, and the role of architectural support (how much and of what kind) needs to be better understood; however, overall the field needs to move to a more integrated approach in systems, applications and tools if its performance gap with hardware cache coherence is to be better understood and reduced.

We believe that the most productive way to improve protocols and systems is by understanding applications and their protocol interactions and performance bottlenecks. The maturity of protocols has opened up several important higher-level areas in which research should now be conducted. First, we should understand how applications may be structured to interact better with software shared memory systems, and programmed so that they port well in performance across software and hardware shared memory. Second, we should try to understand the tradeoffs between relaxation of consistency models and the granularity of coherence; in particular, to understand which of fine-grained and page-grained software shared memory is likely to be more successful as systems evolve into the future. And finally, an important new area for protocols (and applications) is using software shared memory to extend the shared address space programming model across clusters of cache-coherent multiprocessors. Despite the years of work on protocols, there is clearly a lot of exciting work left to do in understanding the role and potential of software shared memory as an alternative to hardware cache coherence.

References

[1] S. V. Adve, A. L. Cox, S. Dwarkadas, R. Rajamony, and

W. Zwaenepoel. A Comparison of Entry Consistency and Lazy Release Consistency Implementation. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[2] C. Amza, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. Software DSM Protocols that Adapt between Single Writer and Multiple Writer. In *The 3rd IEEE Symposium on High-Performance Computer Architecture*, 1997.

[3] A.R. Lebeck B. Falsafi, S.K. Reinhart, I. Schoinas, M.D. Hill, J.R. Larus, A. Rogers, and D.A. Wood. Application-Specific Protocols for User-Level Shared Memory. In *Supercomputing '94*, 1994.

[4] B.N. Bershad and M.J. Zekauskas. Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors. Technical Report CMU-CS-91-170, Carnegie Mellon University, September 1991.

[5] R. Bianchini, L.I Kontothanassis, R. Pinto, M. De Maria, M. Abud, and C.L. Amorim. Hiding Communication Latency and Coherence Overhead in Software DSMs. In *The 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[6] A. Bilas, L. Iftode, D. Martin, and J.P. Singh. Shared Virtual Memory Across SMP Nodes Using Automatic Update: Protocols and Performance. Technical Report TR-517-96, Princeton, NJ, March 1996.

[7] Angelos Bilas and Jaswinder Pal Singh. The Effects of communication Parameters on End Performance of Shared Virtual Memory Clusters. In *Proceedings of Supercomputing'97*, November 1997.

[8] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. A Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 142–153, April 1994.

[9] J. B. Carter, J. K. Bennett, and Willy Zwaenepoel. Techniques for Reducing Consistency-Related Communication in Distributed Shared-Memory Systems. *ACM Transactions on Computer Systems*, 13(3):205–244, August 1995.

[10] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[11] M. Dubois, J.C. Wang, L.A. Barroso, K. Lee, and Y-S Chen. Delayed Consistency and Its Effects on the Miss Rate of Parallel Programs. In *Supercomputing '91*, pages 197–206, 1991.

[12] S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. An Integrated Compile-Time/Run-Time Software Distributed Shared Memory Systems. In *ASPLOS-VII*, 1996.

[13] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy. SoftFLASH: Analyzing the Performance of Clustered Distributed Virtual Shared Memory. In *The 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[14] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 15–26, May 1990.

[15] Richard Gillett. Memory Channel Network for PCI. In *Proceedings of Hot Interconnects '95 Symposium*, August 1995.

[16] L. Iftode, C. Dubnicki, E. W. Felten, and Kai Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[17] L. Iftode, J. P. Singh, and Kai Li. Understanding Application Performance on Shared Virtual Memory. In *Proceedings of the 23rd Annual Symposium on Computer Architecture*, May 1996.

[18] L. Iftode, J.P. Singh, and K. Li. Scope Consistency: a Bridge Between Release Consistency and Entry Consistency. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1996.

[19] Dongming Jiang, Hongzhang Shan, and Jaswinder Pal Singh. Application Restructuring and Performance Portability Across Shared Virtual Memory and Hardware-Coherent Multiprocessors. In *Proceedings of the 6th ACM Symposium on Principles and Practice of Parallel Programming*, June 1997.

[20] M. Karlsson and P. Stenstrom. Performance Evaluation of Cluster-Based Multiprocessor Built from ATM Switches and Bus-Based Multiprocessor Servers. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[21] P. Keleher. Is There No Place Like Home. In *Submitted for publication*, 1997.

[22] P. Keleher, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter USENIX Conference*, pages 115–132, January 1994.

[23] P. Keleher, A.L. Cox, S. Dwarkadas, and W. Zwaenepole. An Evaluation of Software-Based Release Consistent Protocols.

[24] P. Keleher, A.L. Cox, and W. Zwaenepoel. Lazy Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 13–21, May 1992.

[25] P.J. Keleher. The Relative Importance of Concurrent Writers and Weak Consistency Models. In *Proceedings of the IEEE COMPCON '96 Conference*, February 1996.

[26] L. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Ciernak, S. Parthasarathy, W. Meira Jr., S. Dwarkadas, and M. Scott. VM-based Shared Memory on Low-Latency, Remote-Memory-Access Networks. In *ISCA24*, 1997.

[27] L. I. Kontothanassis and M. L. Scott. Using Memory-Mapped Network Interfaces to Improve the Performance of Distributed Shared Memory. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[28] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocessor Programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[29] K. Li. *Shared Virtual Memory on Loosely-coupled Multiprocessors*. PhD thesis, Yale University, October 1986. Tech Report YALEU-RR-492.

[30] R. Samanta, L. Iftode, J.P. Singh, and Kai Li. Home-based SVM Protocols for SMPs. Technical report tr-535-96, Princeton University, Princeton, November 1996.

[31] D. Scales and K. Gharachorloo. Towards Transparent and Efficient Software Distributed Shared Memory. In *SOSP-16*, 1997.

[32] D.J. Scales, K. Gharachorloo, and C.A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *The 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[33] I. Schoinas, B. Falsafi, A.R. Lebeck, S.K. Reinhardt, J.R. Larus, and D.A. Wood. Fine-grain Access for Distributed Shared Memory. In *The 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–306, October 1994.

[34] E. Speight and J.K. Bennett. Brazos: A Third Generation DSM Systems. In *USENIX Workshop on Windows-NT*, 1997.

[35] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *SOSP-16*, 1997.

[36] D. Yeung, J. Kubiatowicz, and A. Agarwal. MGS: A Multigrain Shared Memory System. In *Proceedings of the 23rd Annual Symposium on Computer Architecture*, May 1996.

[37] Y. Zhou, L. Iftode, and K. Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. In *Proceedings of the Operating Systems Design and Implementation Symposium*, October 1996.

[38] Y. Zhou, L. Iftode, J.P. Singh, K.Li, B.R. Toonen, I.Schoinas, M.D. Hill, and D.A. Wood. Relaxed Consistency and Coherence Granularity in DSM Systems: A Performance Evaluation. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1997.